

(19)日本国特許庁 (J P)

(12) 公 開 特 許 公 報 (A)

(11)特許出願公開番号

特開平10-116302

(43)公開日 平成10年(1998) 5月6日

(51)Int.Cl.⁶

G 0 6 F 17/50

識別記号

F I

G 0 6 F 15/60

6 5 4 A

6 5 4 D

6 5 4 G

審査請求 未請求 請求項の数15 O L (全 19 頁)

(21)出願番号 特願平9-249152

(22)出願日 平成9年(1997) 9月12日

(31)優先権主張番号 9 6 1 9 0 9 6 . 2

(32)優先日 1996年 9月12日

(33)優先権主張国 イギリス (G B)

(71)出願人 000005049

シャープ株式会社

大阪府大阪市阿倍野区長池町22番22号

(72)発明者 アンドリュー ケイ

イギリス国 オーエックス4 1エイチエ

イ オックスフォード, ハースト スト

リート 99

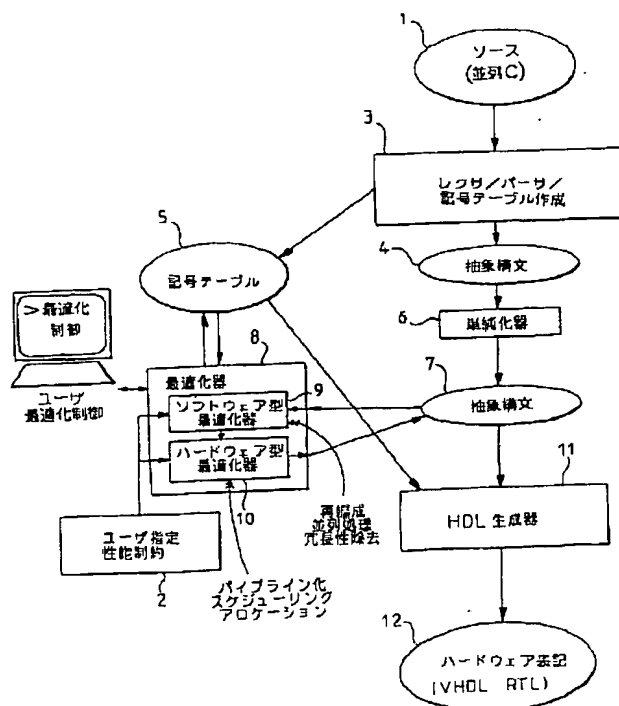
(74)代理人 弁理士 山本 秀策

(54)【発明の名称】 集積回路の設計方法及びそれによって設計された集積回路

(57)【要約】

【課題】 高レベル言語で記述されたソフトウェアレベルからハードウェアレベルへの変換時間を短縮し、ハードウェア開発の効率を向上させる集積回路の設計方法を提供する。

【解決手段】 並列処理及び同期通信をサポートするプログラミング言語でその機能を定義することによって、集積回路が設計される。得られたソースコード(1)が、コンパイラに与えられる。コンパイラは、集積回路の外部通信の順番を変えずに同期通信のタイミングを変える最適化器モジュール(8)を含む。コンパイラは、集積回路の回路構成を表す出力コード(12)を生成する。出力コード(12)は、統合ツール、及び集積回路の製造における後工程にも提供され得る。



【特許請求の範囲】

【請求項 1】 集積回路の機能を並列処理及び同期通信をサポートするプログラミング言語で定義するステップと、

該集積回路の外部通信の順番を変えずに同期通信のタイミングを変えるように構成されたコンパイラを適用して、該集積回路の回路構成を表す出力コードを生成するステップと、を包含する、集積回路の設計方法。

【請求項 2】 前記同期通信がハンドシェークを含む、請求項 1 に記載の集積回路の設計方法。

【請求項 3】 前記コンパイラが、抽象構文木及び記号テーブルを形成するように構成されている、請求項 1 或いは 2 に記載の集積回路の設計方法。

【請求項 4】 前記コンパイラが、前記抽象構文木を単純化するソフトウェア最適化器を含む、請求項 3 に記載の集積回路の設計方法。

【請求項 5】 前記ソフトウェア最適化器が、各コンポーネント部分に変数が 1 つ存在するように複合データ構造をコンポーネント部分に変換するように構成されている、請求項 4 に記載の集積回路の設計方法。

【請求項 6】 前記ソフトウェア最適化器が、未使用変数を除去するように構成されている、請求項 4 或いは 5 に記載の集積回路の設計方法。

【請求項 7】 前記ソフトウェア最適化器が、ループ外の共通演算子を移動させるように構成されている、請求項 4 から 6 のいずれか一つに記載の集積回路の設計方法。

【請求項 8】 前記コンパイラが、前記出力コードによって表記されるハードウェアインプリメンテーションを最適化するハードウェア最適化器を含む、請求項 1 から 7 のいずれか一つに記載の集積回路の設計方法。

【請求項 9】 前記ハードウェア最適化器が、スケジューリング及びアロケーションを行うように構成されている、請求項 8 に記載の集積回路の設計方法。

【請求項 10】 前記コンパイラが、少なくとも 1 つの所定の性能パラメータが達成されたときに最適化を終了するように構成されている、請求項 4 から 9 のいずれか一つに記載の集積回路の設計方法。

【請求項 11】 前記少なくとも 1 つの所定の性能パラメータが、集積回路の最大面積を含む、請求項 10 に記載の集積回路の設計方法。

【請求項 12】 前記少なくとも 1 つの所定の性能パラメータが、集積回路の最小処理速度を含む、請求項 10 或いは 11 に記載の集積回路の設計方法。

【請求項 13】 前記少なくとも 1 つの所定の性能パラメータが、最大消費パワーを含む、請求項 10 から 12 のいずれか一つに記載の集積回路の設計方法。

【請求項 14】 前記出力コードで定義される構成を実行する回路構成を表すレジスタ転送レベルコードを生成するステップをさらに含む、請求項 1 から 13 のいずれ

か一つに記載の集積回路の設計方法。

【請求項 15】 請求項 1 から 14 のいずれか一つに記載の方法によって設計された集積回路。

【発明の詳細な説明】

【0001】

【発明の属する技術分野】本発明は、集積回路の設計方法に関する。本発明はまた、そのような方法によって設計された集積回路に関する。

【0002】

10 【従来の技術】大規模集積 (LSI) 回路のデザインは、例えば AND、OR、NOT、FLIP-FLOP 等の 2 進機能を実行するゲートの集合体と、それらのゲートの相互接続に関する仕様と、を含む。その後、デザインを適切な技術での製造に適した形式に変換するために、レイアウトツールが使用され得る。

【0003】このようなデザインを作成する公知の技術では、「概略的な獲得 (schematic capture)」の名で知られる方法が用いられる。この技術によれば、ユーザは、グラフィックソフトウェアツールを用いて、ライブラリから得た各論理ゲート或いはゲートの集合体を配置し、コンピュータのマウスを用いて配線を「描く」ことによってこれらのゲートを相互接続することができる。その後、例えばゲートを除去或いは単純化することによって、回路の全体機能を変えずに得られた回路を最適化し、これをレイアウト及び製造工程に出すことができる。しかし、設計者は、全て或いは殆ど全てのゲート或いはゲートの集合体についてのタイミング及び論理を考慮しなければならない。従って、この技術は、大規模な設計に使用することが難しく、またエラーを生じ易い。

20 【0004】別の公知の技術では、設計者が、LSI 回路の記述をハードウェア記述言語 (HDL) で書く。HDL の各ステートメントは最終デザインにおける数個のゲートに対応するので、最終デザインにおける論理の複雑さに比べれば入力ソースコードは比較的短い。従って、設計者の生産性が向上する。公知の HDL には、IEEE Standard VHDL Language Reference Manual, IEEE Std 1076-1993, IEEE, New York, 1993 に開示される VHDL、及び D. E. Thomas and P. R. Moorby により The Verilog Hardware Description Language, Kluwer Academic 1995 に開示される Verilog がある。このような言語を S. Carlson により Introduction to HDL-Based Design Using VHDL, Synopsys Inc., CA, 1991 (文献 1) に開示されるような適切な統合ツールとともに用いることにより、デザインを回路構成に変換する。

40 【0005】このような HDL を用いた統合技術を利用して新たな LSI 回路を設計する際には、その回路の挙動についてのアルゴリズムが、ソフトウェア技術者によって、C 言語として知られるような適切な高レベル言語でキャプチャされる。その後、例えば C 言語で書かれた「テストハーネス」を用いて、アルゴリズムの挙動が

3

正しいかどうかを調べるテストを行う。テストハーネスは、回路シミュレータ或いはエミュレータを用いて、その回路設計がテストされ得る環境を記述する。ディスク或いはランダムアクセスメモリ (RAM) に格納された回路に対しては、ベクトルとして知られている入力セットを利用したテストをコンパイルして且つ実行するために、標準的なコンパイラを有するワークステーションが用いられる。

【0006】次のステップでは、文献1に開示されるVHDLレジスタ転送レベル(RTL)のような、ハードウェアの統合及びシミュレーションにより適した言語に、ハードウェア技術者がCコードを書き換える。この時点では、どの種のアーキテクチャを用いるか、データをパイプライン化するかどうか、外部に対する回路インターフェースをどのようにするか、及び各ストラクチャに何ビット分の格納場所をアロケートするのか等、多数の設計上の選択肢が存在する。典型的には、VHDLバージョンは、オリジナルのCバージョンに比べて、大きさが1桁大きくなる。

【0007】CバージョンとHDLバージョンとの間に直接的な結び付きがないので、HDLの記述にエラーが生じることがあり、そのために、この段階でテストを行うことが必要不可欠である場合が多い。デザインがテストされ得る前には、例えばVHDLなどで新しいテストハーネスを書かなければならない。このハーネスもまた、C言語で書かれたハーネスに比べて1桁大きくなる場合が多い。VHDLバージョンを綿密にテストした後には、上記のような適切な統合ツールを用いて、回路に変換することができる。しかし、回路に統合され得るVHDL構成のセットは、VHDL言語全体のサイズと比べて比較的に小さい。また、タイミング及びアーキテクチャに関する決定の殆どは、ユーザによって明示的に注釈されなければならない。従って、ユーザは個々の言語構成がどのように統合されるかに関してかなり詳細な知識を持っていなければならない。この知識は、異なる統合ツール間では異なる。

【0008】この時点で、統合された回路が、所望のデザインに対して遅過ぎたり大き過ぎたりすることを発見することができる。そのような場合、HDLを調節してデザインを指定の範囲内に戻すことができる場合もあるが、それができない場合には、C言語で書かれた新たなアルゴリズムを試作する必要がある、設計時間が長くなってしまふ。

【0009】例えばD. Gajski, N. Dutt, A. Wu and S. LinによってHigh-Level Synthesis, Introduction to Chip and System Design, Kluwer, 1992 (文献2)に開示されているような高レベルハードウェア設計言語を提供するために、HDLの抽象化レベルの向上が図られている。その一例に、Synopsys On-Line documentation 3.2 b (CDROM format), Synopsys Inc., CA, 1995に開示さ

4

れているシノプシス・ビヘイビア・コンパイラがある。このコンパイラは、「挙動性(behavioural)」VHDLで書かれたソースコードを受け取り、比較的の低レベルな統合可能VHDLの出力を生成する。この入力言語は、標準的な統合可能サブセットよりも広いフルVHDL言語のサブセットから派生する。コンパイラは、そのデザインのアーキテクチャを選択し、マイクロプロセッサコアとしてそれをモデリングし、回路全体の速度要件に見合う十分なハードウェアが利用可能であるようにする。このコンパイラは最適化を提供し、文献2に開示されるようなスケジューリング及びアロケーション方式アルゴリズムによって速度と面積とのトレードオフを行う。

【0010】しかし、依然としてユーザは、クロック端がどこで生じるのかを注釈することによってタイミング情報を提供し、且つ、どのクロックサイクルで入力及び出力データが利用可能でなければならないのかを知っている必要がある。このような理由から、このシステムを使用しようとする設計者には、ハードウェアに関する相応な知識が要求される。また、得られるハードウェア記述は、オリジナルの挙動性VHDL記述とは挙動が異なるので、2つの異なるテストハーネスが必要となり得る。さらに、このシステムは、そのタイミング要件への依存性のために、アルゴリズムの試作には適さない。なぜなら、これらは、現在はクロックサイクルレベルであり、サブクロックレベルではないからである。

【0011】他の公知のコンパイラとしては、I. Page and W. LuckによりCompiling Occam into FPGAs, 271-283, Abingdon EE & CS books, 1991に開示されているようなHandelコンパイラ及びHandel-Cコンパイラがある。Handelコンパイラは、例えばInmos, The Occam 2 Programming Manual, Prentice-Hall International, 1988に開示されるような、Occamの名で知られる言語で書かれたソースコードを受け取る。OccamはCに似た言語であるが、並列処理及び名前付きチャネルを介した同期2点間通信を表現するための余分な構成を有する。Handel-Cコンパイラも殆ど同じであるが、ソース言語が若干異なっており、C言語に慣れたプログラマにとってより馴染み易いものとなっている。

【0012】コンパイラが並列構成を提供するので、プログラマは、設計上の問題に対する可能な解決策として、並列アルゴリズムを考えることができる。プログラマがどのサイクルで開始しようともメッセージが全く失われないことを確実にするために、周知のタイプの単純な「ハンドシェーク」技術によって同期通信が行われる。従って、送信側及び受信側の両者は、通信が完了するのを待ってからでなければ、続行することができない。言語によってこの制約が課せられるので、プログラマが通信イベントを再スケジューリングする際の自由度が増す。例えば、プログラマが、c1及びc2と名付け

られたチャネルを介してそれぞれ10及び23という値の送信を要求する場合、受信プロセスが適切に書かれているとすれば、上記データは、どの順番でも、並列にも送信され得て、或いは、sendコマンドの前とsendコマンドの間とに任意の遅延を伴っても送信され得る。このための疑似コードの一例は、

【0013】

【数1】

seq[send(c1,10);send(c2,23);]

OR seq[send(c2,23);send(c1,10);]

OR par[send(c1,10);send(c2,23);]

OR seq[delay(x);send(c1,10);delay(y);send(c2,23);]

【0014】のようになる。

【0015】ハンドシェイクプロトコルは、(どのような形態で実施されていても)受信側が準備状態にあるときにデータアイテムが必ず受信され且つ全く失われないことを確実にする。このようにして、コンパイルされた回路の2つの部分が正確にいつ相互作用するかに関して、幾分の自由度がある。

【0016】しかし、Handelの場合は、(通信以外の)各構成のタイミングの総合的な制御は、プログラマが行う。各構成には、正確なサイクル数が割当てられている(これを、タイムドセマンティクスと呼ぶ)。従って、プログラマは、デザインにおける全ての低レベル並列処理を考慮しなければならない、且つ、コンパイラが各構成をどのようにクロックサイクルに割当てるのかを知っていなければならない。例えば、プログラマは、

【0017】

【数2】

$a := b * c + d * e$

【0018】を指定することができる。しかし、全てのアサインメントに1サイクルしか要しないので、両方の乗算1サイクルで行われる必要がある。これは、2つの乗算器が形成されなければならないことを意味し、面積が大きくなる。また、これらの乗算器が単一のサイクルで動作しなければならないので、クロック速度が遅くなる。

【0019】さらに、主としてタイムドセマンティクスのために、Handelが対応できない重要な構成がいくつかある。そのような構成には、あるアレイ(RAM)を2回参照するアサインメント(なぜなら、これは高価なデュアルポートRAMを示唆する)、ファンクションコールを含む表現、ならびにパラメータ付き関数が含まれる。

【0020】

【発明が解決しようとする課題】タイムドソース言語セマンティクスを用いる公知の言語は、オリジナルのソースコードにおける暗示的なタイミングに従ってしまう。

従って、コンパイラがプログラムの実行に要するサイクル数を変えるような最適化を行うことができないことがあって、オリジナルソースコードにおけるタイミング挙動をエンコードするのがユーザの責任になる。従って、タイムドセマンティクスを利用する場合には、設計者自身が最適化を行わなければならない、コンパイラがそれを助けることはできないので、設計時間の点で問題がある。

【0021】本発明は、上記課題を克服するためになされたものであって、その目的は、(1)高レベル言語で記述されたソフトウェアレベルからハードウェアレベルへの変換時間を短縮し、ハードウェア開発の効率を向上させる集積回路の設計方法を提供すること、及び(2)そのような設計方法に従って設計された集積回路を提供すること、である。

【0022】

【課題を解決するための手段】本発明の第1の局面によれば、集積回路の機能を並列処理及び同期通信をサポートするプログラミング言語で定義するステップと、該集積回路の外部通信の順番を変えずに同期通信のタイミングを変えるように構成されたコンパイラを適用して、該集積回路の回路構成を表す出力コードを生成するステップと、を包含する集積回路の設計方法が提供され、そのことによって上記目的が達成される。

【0023】前記同期通信は、ハンドシェイクを含み得る。

【0024】前記コンパイラは、抽象構文木及び記号テーブルを形成するように構成され得る。前記コンパイラは、前記抽象構文木を単純化するソフトウェア最適化器を含み得る。前記ソフトウェア最適化器は、各コンポーネント部分に変数が1つ存在するように複合データ構造をコンポーネント部分に変換するように構成され得る。前記ソフトウェア最適化器は、未使用変数を除去するように構成され得る。前記ソフトウェア最適化器は、ループ外の共通演算子を移動させるように構成され得る。

【0025】前記コンパイラは、前記出力コードによって表記されるハードウェアインプリメンテーションを最適化するハードウェア最適化器を含み得る。前記ハードウェア最適化器は、スケジューリング及びアロケーションを行うように構成され得る。

【0026】前記コンパイラは、少なくとも1つの所定の性能パラメータが達成されたときに最適化を終了するように構成され得る。前記少なくとも1つの所定の性能パラメータは、集積回路の最大面積を含み得る。前記少なくとも1つの所定の性能パラメータは、集積回路の最小処理速度を含み得る。前記少なくとも1つの所定の性能パラメータは、最大消費パワーを含み得る。

【0027】該方法は、前記出力コードで定義される構成を実行する回路構成を表すレジスタ転送レベルコードを生成するステップをさらに含み得る。

【0028】本発明の第2の局面によれば、本発明の第1の局面による方法によって設計された集積回路が提供され、そのことによって上記目的が達成される。

【0029】以下に、本発明の作用を説明する。

【0030】ハンドシェーク等の通信プロトコルを高レベルの最適化と共に使用すれば、コンパイラは効率的なインプリメンテーションをもたらす得るので、通信を抽象的に表現する際により高い自由度が設計者にもたらされる。入力言語は、高レベルで且つプログラマに馴染み易いものであり得るとともに、ハードウェアにおいて認識可能な表記を有する重要な構成の殆どをサポートし得る。言語は、並列処理及び2点間通信を表現し得るが、タイムドセマンティクスを持たない。コンパイラは、例えばソースコードレベルに近い比較的高レベルで最適化を行うことができるとともに、HDLを出力することができる。従って、低レベルの統合、最適化及びハードウェアマッピングは、業界標準ツールを用いて行うことができる。並列構成及び2点間通信を伴うC言語に類似したソフトウェア言語を用いたデザインの機能は、効率的なLSI設計のために自動的に或いは半自動的にHDLに変換される。アンタイムドソース言語セマンティクスを使用すれば、コンパイラによって、オリジナルソース言語の仕様に従いながらデザインのタイミングを向上させる最適化を行うことが可能になる。

【0031】

【発明の実施の形態】以下に、添付の図面を参照しながら本発明の実施例を説明する。

【0032】図1に示されるコンパイラは、「並列C言語」の名で知られる高レベル言語で書かれたソースコード1を受け取る。この言語は、構成とハンドシェーク2点間通信プリミティブとを含み、例えば最終設計のコスト及び性能に関するユーザ指定性能制約（参照符号2で図示）を指定する。コンパイラは、モジュール3を有する。モジュール3は、入力ソースコードの構文解析及びチェックを行うことにより、中間抽象構文木（AST）表記4及び記号テーブル5を作成する。記号テーブル5は、ソースコードによって宣言された名前及びオブジェクトを記録するものである。A. V. Aho and J. D. UllmanによりPrinciples of Compiler Design, Addison-Wesley, 1977（文献3）第197～244頁に、これに適したレキサ及びパーサが開示されている。中間構文は、シーケンシャル、並列、及びループ構造のための表記を有し、また、演算のスケジューリング及びアロケーションを表すために用いられる特定の注釈を有する。

【0033】ASTは、単純化器モジュール6に与えられる。単純化器モジュール6は、全ての不体裁な構成を、よりシンプルな構成に書き直す。特に、後のステージをより単純にコード化できるように、単純化器モジュール6からの出力はフル抽象構文のサブセットだけが使用されている。例えば、単純化器モジュール6は、複合

データ構造を、各コンポーネントに変数が1つであるコンポーネント部分に分解する。

【0034】単純化されたAST7は、最適化器モジュール8に与えられる。最適化器モジュール8は、ソフトウェア最適化器モジュール9とその次にあるハードウェア最適化器モジュール10とを含んでいる。ソフトウェア最適化器モジュール9は、例えば文献3の第406～517頁に開示されるような、不使用変数の除去及びループ外の共通処理の移動のようなソフトウェア最適化技術を用いて、単純化されたAST7を最適化する。ハンドシェークのようなプロトコルを用いて通信を行うので、転送中にデータが失われることがない。従って、デザインにおける最終タイミングに影響を及ぼし得るが、そのような通信の時間的順序は変えないように、単純化されたAST内で通信の移動を行うことができる。並列処理を許容する言語でソースコード1が書かれているため、ソフトウェア最適化器モジュール9は、性能制約2を満たすように計算をパイプライン化する等の手段を導入することができる。

【0035】全ての最適化が設計の向上に貢献するように、推定関数が用いられる。推定関数を用いて、ゲート数、回路面積、回路速度、待ち時間、スループット、消費パワー、リソース要件などを推定することができる。ソフトウェア最適化が完了すると、ハードウェア最適化器モジュール10は、ハードウェアターゲットに固有の最適化を行う。文献2の第137～296頁に、適切なハードウェア最適化器が開示されている。回路面積及びゲート数はハードウェアの作成における重要な考慮点であるので、上記の最適化は、可能な限りハードウェアが再利用され得るように、タイミングを考慮して設計される。このプロセスには、パイプライン化、スケジューリング、及びアロケーションの技術が含まれ、各最適化に対して再び推定関数を行うことにより、確実に改善がなされているようにする。モジュール10による最適化の結果によって、各変数及び演算子の最適なハードウェア表記についての追加情報が、記号テーブル5に追加される。例えば、初期化された後に書込みが全く行われないアレイは、読出し／書込みアレイに必要なRAMよりも安価なリードオンリーメモリ（ROM）によって表すことができる。

【0036】デザインが、ユーザによって指定された性能及びコスト制約2を満たしていることが推定関数によって示されたとき、モジュール9及び10による最適化が終了し得る。これらの制約を満たすことが不可能な場合、ユーザにメッセージが与えられ得る。さらに、特定の最適化を選択する、或いは、最適化のパラメータを与えることによって、最適化器モジュール8に命令を与えるためのユーザインタラクションがさらに設けられても良い。この最適化は、適切な環境下におけるデザインの機能性には影響を与えないが、これによって、異なる性

能或いはコストを達成し得る。

【0037】最適化されたAST及び改変された記号テーブルは、HDL生成器モジュール11に与えられる。HDL生成器モジュール11は、ASTを詳細に検討し、そして、記号テーブル5に集められた情報を用いて各構成についてのハードウェア表記を作成する。このようにして、生成器モジュール11は、VHDLRTL等の適切な言語で書かれたハードウェア表記12を提供する。その後、業界標準ツールを用いて、RTLをLSI回路に変換することができる。

【0038】モジュール8が行い得る最適化の種類の3つの例を、以下に説明する。

【0039】(例1) この例は、ソフトウェア式の最適化によってアルゴリズム性能を向上させる1つの方法を示す。

【0040】

【数3】

```
a := 3
b := 1
while (true)
    b := b + (a * a)
send (ch, b)
```

【0041】というコードを考える。この例に適用可能な最適化がいくつかある。まず、 $a * a$ はループの反復の度に計算されるが、 a の値はループ内で変わらないので、答は常に同じである。ループ開始前に行う1回の計算と1回の一時的変数の代入とによって、これを置換し得る。これは、ハードウェア上では、ループの実行中に乗算器を開放して他の場所での乗算器の使用を可能にするとともに、乗算器が2サイクル以上のサイクルを要求する場合にはループ待ち時間を短縮する可能性もある。これが可能であるのは、`send` コマンドがタイミングに依存しないからであり、

【0042】

【数4】

```
a := 3
b := 1
tmp := a*a
while (true)
    b := b+tmp
send (ch, b)
```

【0043】のようになる。

【0044】2番目の最適化は、 $a = 3$ 及び $tmp = 9$ とすることである。従って、このプログラムは、

【0045】

【数5】

```
10
a := 3
b := 1
while (true)
    b := b+9
send (ch, b)
```

【0046】のように書くことができる。

【0047】次に、 a は一度も読み出されないで、

【0048】

10 【数6】

```
b := 1
while (true)
    b := b+9
send (ch, b)
```

【0049】のように、 a を取ってしまうことが可能である。

【0050】(例2) ハードウェア式の最適化の例として、

20 【0051】

【数7】

```
a = b*c + d*e
```

【0052】というアサインメントを考える。

【0053】先に述べたように、公知のコンパイラにおけるタイムドセマンティクスでは、やはり、ソース言語で書かれたこのステートメントは、実行時には、1クロックサイクルで実行されなければならない。これは、2つのフラッシュ乗算器及び1つの加算器を設ける他には、あまり選択の余地がない。結果的に、2つの乗算器のために面積が大きくなり、サイクル時間が遅くなる(フラッシュ乗算器は、典型的に、入力幅に依存して大きな組合せ遅延を有する)。

【0054】しかし、図1のコンパイラには、上記のような制限が全くない。上記の乗算は、

【0055】

【数8】

```
a := b*c
a := a + d*e
```

40

【0056】と示すようにシーケンシャル化することができる。

【0057】次に、

【0058】

【数9】

```
a := sys_mult(b,c)
a := a + sys_mult(d,e)
```

【0059】に示すように、上記乗算を共有乗算関数に代入することが可能である。

【0060】乗算器は、その時の処理の種類に合わせることができる（高速ではあるが大型なパラレル乗算、または、数サイクルを必要とはするが非常に小型で且つクロック速度の速いシーケンシャル乗算）。いずれの場合も、乗算器は1つで十分である。実際の選択は、自動的になされても、或いは、コンピュータのガイダンスによってユーザが行ってもよい。

【0061】（例3）この例では、最終的なデザインの効率を向上させるために、どのようにアサインメントの再スケジューリングを行うかを示す。

【0062】

【数10】

```
a = b*c;
c = x*y;
send (output, a);
b = a+d;
```

【0063】というプログラム例を考える。

【0064】乗算器と加算器とが1つずつある場合、受信側がaの値を受信する準備ができていれば、上記プログラム全体を、

【0065】

【数11】

```
a = b*c;
par
{
c = x*y;
send (output, a);
b = a+d;
}
```

【0066】のように、2サイクルに圧縮できる。

【0067】公知のコンパイラでは、通信をこのように再スケジューリングすることができないので、この最適化を行うことはできない。例えば、タイムドセマンティクスを有するコンパイラでは、ソース言語のタイミング

declaration = ...

| [storage_class] chan [type]identifier[,identifier]*;

【0077】に示す宣言の構文の拡張を用いて宣言される。

【0078】例えば、

【0079】

【数15】

```
chan struct comm a, b;
```

【0080】のようにすれば、構造成タイプcommのデータを用いて通信する2つの内部チャンネルa及びbが宣言される。

を変えることはできず、一方、挙動性コンパイラ(Behavioral Compiler)のようなコンパイラでは、通信によって課せられる境界を越えて最適化を行うことはできない。

【0068】あるデザインのソースコード1は、C言語のサブセットにいくつかの追加を含む並列C言語で書かれている。追加分は、以下の通りである。

【0069】(a) 並列処理のための構成: par

par構成は、プログラム内のどこにでも使用することができ、これにより、システムレベルから単一ステートメントレベルまでのあらゆる細分性の並列処理が導入され得る。この構文は、

【0070】

【数12】

```
statement = ...
| par {[statement]*}
```

【0071】に示すように、通常のCステートメント構文を拡張する。

【0072】例えば、

【0073】

【数13】

```
par {
func1(x,y);
func2(x+y,z);
}
```

【0074】のようにして、2つのファンクションコールを並列に実行する。

30 【0075】(b) 所与のタイプの同期チャンネル

これらのチャンネルにより、1つのparにおけるブランチ間での通信、及び、(C言語の通常の外部キーワードとともに用いられる場合に)同期回路とその周辺との間での通信が可能になる。内部チャンネルは、

【0076】

【数14】

【0081】チャンネルは単方向性であるので、周辺との通信を行うには、そのプロセスが、任意の共有チャンネルの送信端或いは受信端のいずれを有しているのかが分かっている必要がある。これは、キーワードchan in及びchan outによって区別されるので、全ての外部チャンネル宣言において、これらのキーワードを使用しなければならない。

【0082】

【数16】

13
declaration = ...

```
| [storage_class] chanin [type] identifier [identifier]*;
| [storage_class] chanout [type] identifier [,identifier]*;
```

【0083】例えば、

【0084】

【数17】

```
extern chanin int from_env;
extern chanout int to_env;
```

【0085】のようにして、16ビットの整数で周辺と通信するチャンネル `from_env` 及び `to_env` を宣言する。

【0086】(c) プリミティブ `send (chan, val)` 及び `receive (chan)` `send (chan, val)` は、チャンネル `chan` を介して値 `val` を送信する。`receive (chan)` は、チャンネル `chan` を介して値を受信するものであり、表現を作成する際に使用できる。各チャンネルは2点間方式でデータの通信を行い、通信を行っている2つの処理はそれぞれ、通信が完了するのを待ってからでなければ続行できない。さらに、チャンネル `chan` を介して送信されるのを待っているデータが存在するときに真となる関数 `ready (chan)` がある。

【0087】

【数18】

```
statement = ...
| send (identifier, expression);
expression = ...
| receive (identifier)
| ready (identifier)
```

【0088】次の例は一对のプロセスを示し、一方のプロセスは、整数を生成してそれを他方のプロセスに（チャンネル `ch` を用いて）送信し、この他方のプロセスは、受信した整数を加算する。

【0089】

【数19】

```
chan int ch;
par {
    {
        int i = 0;
        while (1)
            send (ch, i++);
    }
    {
        int tot = 0;
        while (1)
            tot += receive(ch);
    }
}
```

10

20

【0090】(d) 所与のビット幅の整数型のセットこれは、どのような数値精度が要求された場合でも、効率的な回路が形成できるようにするためのものである。このために、`#e` が含まれるように、型修飾子のセットが拡張される。ここで、`e` は、`e` の値に等しい幅を示す定数表現である。

【0091】

【数20】

```
type_modifier = ...
| #constant_expression
```

30

【0092】例えば、

【0093】

【数21】

```
chan unsigned#7 c;
```

【0094】のようにして、「符号無し7ビット整数」型の `c` と呼ばれるチャンネルを宣言する。

【0095】(e) ビット操作を行う効率的な回路を構築するためのビット選択及びビット連結演算子記号 `@` は、連結を表す。「`grab`」演算子 (`<-` と書く) は、表現 `e`、及び定数ビット位置 `b1` . . . `bn` のリストを要する。この演算子が評価されると、`eb1` . . . `ebn` の `n` ビットの結果が返される。但し、`ei` は、`e` の `i` 番目のビットである。

【0096】

【数22】

40

15
expression = ...

| expression @ expression
| expression < _{constant_expression[constant_expression]*}

16

【0097】例えば、3ビット2進数では $5_{10}=101_2$ であり、 $7_{10}=111_2$ である。従って、6ビット2進数では、 $5_{10} @ 7_{10}=101111_2=47_{10}$ である。 47_{10} から上位4ビットを選択すると、表現 $47_{10} < \{5, 4, 3, 2\}$ は、値 $10111_2=11_{10}$ を生成する。

【0098】入力言語の標準C部分は、if、while、switch、blocks、functions等の全ての制御特徴、ならびに、ポインタを除く演算及びデータ操作の殆ど全てを有する。しかし、アレイのインデックスを用いてポインタを真似ることは可能である。回路の外部のRAM或いはROMコンポーネントであると仮定される「外部アレイ」を除いて、アレイは、統合回路内の専用ロジックとして実現される。

【0099】上記言語におけるC言語部分のセマンティクスは、C言語のセマンティクス（つまり、expressions、assignment、if、while、for、break等）と類似している。par及びチャンネル通信のセマンティクスは、上記INMOSの文献に開示されるOccamのセマンティクス、及び、C.A.R. HoareによってCommunication Sequential Processes, International Series in Computer Science, Prentice-Hall, 1985に開示されるCSPのセマンティクスと類似している。Occamの用法ルールに類似する用法ルールがある。2つの異なる並列コンポーネントから同一の変数がアクセスされる場合、そのアクセスが全てリードオンリーでない限り、挙動は不確定である。

【0100】先に説明したように、ソースコードがファイルに入力された後、図1の3においてコンパイラはコンパイルを開始し、標準的な構文解析技術を用いてソースコードを構文解析して、デザインの構造やサブ構造などを記録する抽象構文木4とするとともに、使用される全ての識別子のタイプ及び名前を記録する記号テーブルを作成する。処理が進むと、記号テーブルは、各識別子についての情報を照合することによって様々な変換ステージを互いに関連付ける。

【0101】次の工程は、図1の6において抽象構文木を単純化することである。これを行うのは、生成器11によって、プログラミング特徴の全てをハードウェアに変換することはできないからである。単純化器モジュール6は、それらのサポートされない構成を除去して、HDL生成器モジュール11によってサポートされる等価な構成に置き換える。例えば、生成器モジュールは、標準的なC言語における $a = (b++) + 5$ のように、アサインメントが副次的効果を有することを許可しない。単純化された等価物は、 $a = b + 5$; $b = b + 1$ のよう

になり得る。さらに、単純化器モジュール6は、全ての演算子及び定数の幅と型とを計算し、この情報を構文木に格納する。

【0102】send(ch, R)は、 $ch := R$ のようなアサインメントに単純化される。この表記は、記号テーブルにおけるchの型によって、それが本当はチャンネル送信であることを「知る」。しかし、この表記法の統一性（どのプロトコルが要求される場合でも、デステイネーションは常にアサインメントの左側に書かれる）のために、後のトランスフォーメーションが殆ど例外無く記述される。同様に、 $x := receive(ch)$ は、 $x := ch$ というアサインメントに単純化される。

【0103】この時点で、デザインは、単純化された抽象構文7及び記号テーブル5の組合せで表現される。HDL生成器が処理できない全ての構成を取り去り、最適化を行う必要がある。例えば、外部アレイ(RAM)へのアクセスは、逆の情報がない限り、シングルポートRAMであると想定される。従って、 $mem[i] := mem[j]$ のような表現は、HDL生成器によって正しく処理されない。なぜなら、HDL生成器は、そのメモリへの2つのアクセスを（ほぼ）同時に生成するからである。この表現は、 $local\ t; t := mem[j]; mem[i] := t$; のように書き換えられる。

【0104】標準的な最適化の1つの可能な方法は、ループ内のある計算を反復する必要がない場合に、その計算をループから取り去ってしまうことである。例えば、

【0105】

【数23】

While (x < 10)

$x := x + (y * y)$

【0106】は、

【0107】

【数24】

declare tmp

tmp := y * y

While (x < 10)

$x := x + tmp$

【0108】のように書き換えられる。

【0109】もう1つの可能な方法は、寿命時間が重ならない複数の変数の間でレジスタを共有することである。例えば、

【0110】

【数25】

```
Sequence:
Declare tmp1
    tmp1 := f(x)
    send(ch,tmp1)
Declare tmp2
    tmp2 := g(y)
    send(ch,tmp2)
```

【0111】は、

【0112】

【数26】

```
Sequence:
Declare tmp
    tmp := f(x)
    send(ch,tmp)
    tmp := g(y)
    send(ch,tmp)
```

【0113】のように書き換えられる。

【0114】最適化器モジュール10が、ある特別な場合にしか用いられない構造を見つけた場合、生成器モジュール11がその情報を利用してより簡潔なコードを生成できるように、最適化器モジュール10は、構文木（或いは記号テーブル）にその情報を記録することができる。例えば、あるアレイが定数によって初期化され、その後に全く更新されない場合、より高価なRAMよりも安価なROMとして、そのアレイが実現され得る。

【0115】抽象構文における1つの重要な構成は、アサインメント同期化の形態である。これにより、数個のアサインメントを同時に実行して、時間若しくは格納スペース或いはそれら両方を節減することができる。例えば、`local tmp; tmp := a; a := b; b := tmp;`とする代わりに、これを最適化して、`synch { a := b AND b := a }`とすることが可能である。アサインメントは、レジスタ、チャネル、及びアレイの値の全ての通信を処理する。この構造により、いくつかの有用なアクションを短縮し、短縮しなかった場合よりも時間を短くすることが可能になる。何が同期化され得るかについては制限があり、その制限は、インプリメンテーションに依存する。

【0116】最適化器モジュール10は、さらに根本的なことをすることが可能であり、例えば、シーケンシャルコードを平行に動作させる、或いは、その逆を行う、チャネルを除去して、より弱い形態の同期化に置き換える、インラインに関数を拡張する、乗算を共有乗算関数へのコールにすることにより乗算器を共有する、コード或いは表現の重複部分を共有する、そして、得られる回路の外部挙動が変わらない場合に複雑な表現をパイ

ライン化することができる。無論、速度、面積及びサイクル数は変化し得るが、外部インターフェースは全てハンドシェークを有するので、通信の順序が守られる場合は、このような変化は影響しない。

【0117】最適化は、自動的に適用されてもよいし、ユーザ命令型もしくはユーザ選択型であってもよい。目標は、特定の用途によって決まる指定の面積、パワー、或いは時間の範囲を達成することである。これらの属性は、抽象表記に適用される単純なメトリクスによって推定され得る。

【0118】HDL生成器モジュール11は、洗練された抽象構文木を受け取り、これをHDLによる回路の記述に変換する。この段階では、抽象構文に残された各構成が、良好に特定されたハードウェアインプリメンテーションを有する。その一部を以下に説明する。全般的な技術は、以下に示す重要な点において、公知の技術とは異なっている。

【0119】(1) アサインメントが、より複雑なものであり得るとともに、チャネル通信及びパラメータ化されたファンクションコールを含み得る。そのためには、例えば乗算器にその引数がいつ準備状態にあるかが知られるように、表現の部分間のより複雑なプロトコルが要求される。

【0120】(2) コンパイラが、ネットリストではなくHDLを生成するので、幾分かの選択の余地がある。具体的には、設計チェーンにおけるより低い統合ツールまで、状態マシン及びレジスタを実行する方法に、選択の余地がある。

【0121】基本的なスキームは、抽象構文内の制御ステートメントからの状態マシンとして、制御パスを統合することである。殆どの状態において、何らかの計算が行われる。状態マシンは、計算を初期化し、その計算が完了するのを待ってから次のステージに進む。例えばIFの場合、次の状態の位置は、計算された値に依存する。1つの状態マシンが、その後に同時に実行される他の1セットの状態マシンを起動し得るようにすることによって、並列処理が行われる。

【0122】それぞれの計算が完了するまでにかかり得る時間は未知であるので、例えば、チャネル或いは外部装置との送信或いは受信が行われる場合、データ依存型の計算が行われる場合、或いは、ファンクションコールが行われる場合には、その計算を実行する回路は、計算の完了を信号で知らせるとともに、その値が使用されるのを待ってからその値をディスアサートできなければならない。これは、以下に述べる表現のプロトコルの複雑さを説明する。最適化工程によって抽象構文が十分に単純化されるならば、より単純なプロトコルを用いることも可能であるが、その場合、実行時間が長くなり得る。さらに、統合後のゲートレベルの何らかの最適化は、必要とされないシグナリングの余分なレベルを除去する。

【0123】図2A～図2Cは、制御パス用の基本的なビルディングブロックがどのように形成されるのかを示す図である。ステージは、抽象構文木によって決まる形状に従う。図2Aには、1プロセスを表す状態マシン15が示されている。大きい円16は制御ノードであり、各制御ノードは、1つ或いは1セットのアクションに関連し得る。最も単純なケースにおいて、これらの制御ノードは、抽象構文言語内の基本プロセスに対応するアサインメント或いは通信である。状態マシン15は、そのアクションが完了してからでないと、次の状態に進むことができない。

【0124】図2Bには、コンポーネント状態マシン17及び18のシーケンシャル構造として、シーケンシャル構造が示されており、第1のマシンの終了状態19が第2のマシンの開始状態と1つになっている。

【0125】図2Cには、並列構造が示されている。一つのマスタープロセス20は特別なものであり、通常の方法によって現在のシーケンシャル状態マシンに挿入される。他の全てのスレーブプロセス21は、マスタープロセスが開始するのを待ってから開始する。

【0126】並列部分の終端では、マスタープロセス20は、全てのスレーブプロセス21が終了するのを待ってから、次に進む。その後、各スレーブプロセス21は初期待機状態に戻って、次の起動に備える。実行中は、マスタープロセス及びスレーブプロセスのステータスは同じである。つまり、両者を区別するのは、その開始方法だけである。

【0127】a及びbが幅8で宣言されたものとして、

【0128】

【数27】

Define ID1("a")

Define 1D2("b")

a:=1

b:=a+b

【0129】という抽象構文のフラグメントを考える。

【0130】図3は、このプログラムのために生成され得る可能な回路例を示す。図3の左側には、この例における抽象状態マシンが示されている。各アサインメントにつき1つ、合計2つの中間状態22があり、また、通常の開始状態23及び終了状態24がある。図3の残りの部分は、可能な回路を示す。フリップフロップ25、26、29及び30は、グローバルクロック（不図示）に接続され、立ち上がりエッジで起動する。

【0131】リセット付きD型フリップフロップ25及び26は、「ワンホット」エンコードにおける状態マシンを表す。これは、各フリップフロップが可能な状態の1つを表していることを意味する。フリップフロップが1を有するときには状態はアクティブであり、そうでないときには非アクティブである。プログラムの開始前に

フリップフロップ25及び26を0に設定するために、リセットライン27が必要である。この他のエンコードを用いて状態マシンを表すことも可能であるが、この例が恐らく最も単純である。

【0132】開始パルス28は、1クロックサイクル毎に1状態の割合で、チェーン上を移動する。これは、特別な例である。なぜなら、各アサインメントは1サイクルしか要しないのが仮定されているからである。より複雑な例の場合、関連アクションが完了するまでパルスを待機させるために、何らかの回路機構を生成する必要がある。

【0133】第1の中間状態においては、変数aのためのレジスタ29のイネーブルビットは真に設定されており、これにより、次の立ち上がりクロックエッジにおいて、8ビットの定数値1（2進数で00000001）を格納することが可能になる。

【0134】第2の中間状態においては、変数bのレジスタ30のイネーブルビットは、シングルサイクル加算器31のイネーブルビットと同様に真に設定されている。従って、a及びbの以前の値は、次の立ち上がりクロックエッジが生じたときにbに格納される。

【0135】図4Aは、R表現32（即ち、その値がデータとして要求されている表現）がどのようにインターフェースされるかを示す。R表現の値が要求されると、信号Requestがアサートされる。上記の値が要求されなくなるまでの間、信号Requestを真に保っておかなければならない。その後、この表現はある値を計算し、その値を信号Rvalueとして出力するとともに、その信号Rvalueが有効であることを示すために信号Readyをアサートする。Rvalueが要求されなくなると、入力信号goが、1クロックサイクルだけ真となり、そして信号Requestは偽になる。信号Requestが次に真となるまで、信号Rvalue及びReadyはディスアサートされる。無論、定数及び組合せ表現等の多くの単純な表現の場合、本スキームにおける明らかな複雑さの大部分は、統合中にゲートレベル最適化器によって容易に単純化され得る。

【0136】図4Bにおいては、A+BについてのR表現が、A及びBについてのR表現33及び34、加算器35、及び組合せロジックから構成される。R表現は、定数、単純変数、アレイのリファレンス、チャネル入力、或いは外部メモリからの読出し、ならびに通常の演算的、論理的及びビット的組合せであり得る。Request及びgo信号は、両方のコンポーネント33及び34に一斉送信され、そのRvalueは加算器35に与えられる。この例において、加算器35は組合せ加算器であると仮定されている。複合物のReadyとして、ReadyのブールANDをとる。この回路は、加算器35を必要な関数に変えるだけで、あらゆる

組合せ表現を実行するのに十分なものとなる。実行する処理が組合せではない場合、演算子自体が、適切な方法によって2つのコンポーネントのRreadyを組み合わせてることによって、Rreadyを提供しなければならない。このような接続は、パワー削減のためにも利用され得る。その場合、入力データが有効になるまで、加算器35をオフにする。

【0137】オペランドと演算子入力との間に幾つかのマルチプレクサを挿入し、演算子出力の上にデマルチプレクサを挿入するだけで、1つの加算器（或いは他の演算子）を、いくつかの計算によって共有することが可能になる。最適化モジュール8がコンフリクトが全く無いこと、例えば一度に2つの計算が同一の演算子を使用しようとしていないことを、確かめる必要がある（これは、スケジューリング及びアロケーションと呼ばれる）。

【0138】図5A～図5Cに、その他のR表現を示す。図5Aは、コール・パイ・バリュー関数がどのようにコールされ得るのかを示す。実際のパラメータ（引数）を連結することによって、1つの表現R38が与えられる。この表現は、準備状態になると、プロセスを開始する。このプロセスは、図4A及び図4Bのスレーブプロセスと同様に関数F39を実行する。Fが一度に2回以上起動されないようにするためにFにとって必要な調停は、図5Aには図示されておらず、この単純なロジックは、Fの内部に設けられている。Fからの全ての戻り値は、Rvalue信号を介して発信側に渡され、また、Fが終了すると、Rreadyがアサートされる。go信号は、F及びRに一斉送信される。

【0139】図5Bは、単純変数をどのように実施するのかを示す。値自体はレジスタ（不図示）に格納されており、そのレジスタからの出力は、Rvalue信号を介して、その出力を要求する各R表現に対して利用可能になっている。Request及びgo信号は無視される。この値は常に利用可能であるので、Rreadyはロジック1に固定されている。

【0140】図5Cは、チャンネルがどのように読み出されるのかを示す。チャンネルのtxready信号が真であれば、その表現は準備状態である。ハンドシェイクの最終部は、go信号である。特定のチャンネルから読出しを行う全てのR表現からのgo信号のORをとることにより、そのチャンネルのrxready（受信準備完了）信号を生成する。

【0141】図6A及び図6Bは、どのようにしてL表現（値のデスティネーションを表す表現）を作成するのかを示す。図6Aは、L表現42のための標準的なインターフェースを示す。L表現は、単純変数、アレイのリファレンス、チャンネルの出力、外部メモリへの書き込み、或いはそれらの組合せであり得る。Request信号を用いて、L表現内のあらゆる埋込みR表現（通常

は、アレイのインデックス計算）を開始する。Lrequest信号は、真性L表現を開始し、また、Lvalue信号に有効なデータが存在するときには真に設定される。格納処理の完了準備が整うと、Lready信号が立ち上がる。最後に、表現の環境が準備状態になったとき、1サイクルの間go信号を真にすることによって、リソースの解放を示す。L表現の組合せの場合、2クロックサイクルを要求できるのはL表現の中の1つだけであり、そして、このサブ表現が、処理全体のタイミングを決定する。他の全てのサブ表現のLreadyは、常に真でなければならない。この条件が満たされない場合、プロトコルが失敗し得る。

【0142】図6Bにおいては、アサインメントがどのように構築されるのかを説明するために、インターフェースが用いられている。

【0143】図7A～図7Cは、いくつかの特定のL表現がどのようにしてエンコードされるのかを示す。図7Aは、単純変数（レジスタ）に対する書き込みがどのように行われるのかを示す。書き込みデータを、3状態ドライバ44を介して、その変数用の書き込みバスの上に流す。3状態ドライバ44は、信号goが送信されるとイネーブルされる。これがうまく作動するように、書き込みに要するサイクル数を1とする。そのレジスタの書き込みイネーブル信号をとって、そのレジスタに書き込みを行う全てのL表現に対する全ての書き込みイネーブル信号の論理ORにする。コンフリクトが全く生じ得ないようにするのは、最適化ステージまでである。

【0144】図7Bは、外部メモリへの書き込みがどのように行われるのかを示す。go信号が到達するまでの間、書き込み完了信号を真に保っておかなければならない。やはり、このメモリ装置に関係する全ての書き込みイネーブルのORがとられなければならない。

【0145】図7Cは、チャンネル出力がどのように行われるのかを示す。所与のチャンネルに対するチャンネル出力の全てのL表現が集められる。そのチャンネルのtxready（発信準備完了）は、（このチャンネルに言及する各L表現に1つずつある）部分的txready信号の全てのORである。個々のrxready信号は、チャンネルrxreadyに直接に接続される。

【0146】図8は、if b then P else Qのインプリメンテーションを示す。R表現bからのready信号は、マルチプレクサによって方向づけられる。このマルチプレクサは、bが返した値によって制御される。これにより、状態マシンがPまたはQのどちらを伴って継続するかが選択される。

【0147】図9は、while C (b) do Pのインプリメンテーションを示す。状態マシンは、bの値によって、再びPを実行するか、或いは、Pを実行せずに次に進むように指示される。

【0148】図10A及び図10Bは、リソースの作成

10

20

30

40

50

方法を示す。変数、アレイ、チャネル、或いは関数のそれぞれが、リソースである。HDL生成器モジュール11が構文木を詳細に検討した後の時点では、各リソースは、1つ以上の様々なR表現及びL表現によってアクセスされている。各リソースについて、HDL生成器モジュール11は、これらのR表現及びL表現の「バックエンド」からの信号を用いて、リソースの正しい挙動を規定する適切な回路を作成しなければならない。

【0149】図10Aは、書き込みイネーブルを有するエッジトリガレジスタ45として実施される、単純変数をどのように形成するのかを示す。L表現からの（書き込まれるべき値を持つ）データバスは結合され、書き込みイネーブル信号と一緒にORをとられる。出力（R表現）は比較的簡単であり、必要とされるところにデータがコピーされるだけである。

【0150】図10Bは、どのようにチャネルが形成されるのかを示す。全てのL表現（チャネル出力）は、それぞれデータバスを有する。それらのデータバスは結合され、このチャネルの全てのR表現（チャネル入力）のデータバスにコピーされる。チャネルの読出し箇所はm箇所あり、また、チャネルの書き込み箇所がn箇所あると仮定されている。図10Bの46において、書き込み側のtxready信号のORをとり、これを読出し側に一斉送信する。同様に、図10Bの47において、読出し側のrxready信号のORをとり、これを書き込み側に一斉送信する。

【0151】（例）

【0152】

【数28】

```
void main()

{

    unsigned #8 x;
    for (x=0; x<10; x++)
    {
        x<<=1;
    }

}
```

【0153】というフラグメント例を考える。

【0154】これは、非常に不自然な例である。なぜなら、入力も出力もないからである。しかし、この例は、短くて理解し易い。変数Xは、値0から始まる。その後、変数Xをインクリメントして、これを左に1ビットシフトし、その工程を変数Xが10未満でなくなるまで反復する。その後に、プログラムを終了する。この時点では、出力が多少整理されており、可読性が高められている。

【0155】記号テーブルは、

【0156】

【数29】

ID1 FUNCTION main void → void

ID2 VARIABLE x int#8

【0157】というエントリを有する。

【0158】抽象構文は、

【0159】

10 【数30】

Define ID1("main")

Declare ID2("x")

FOR (x=0(#8); x<10(#8); x++)

x<<=1;

【0160】となる。

【0161】HDL生成器には「for」構成がなく、副次的効果を持つアサインメントは、その副次的効果を明示しなければならない。従って、上記の例は、

20

【0162】

【数31】

Define ID1("main")

Declare ID2 ("x")

x := 0(#8)

While (x < 10(#8))

x := x << 1

x := x + 1(#8)

30 【0163】のように単純化される。

【0164】最適化器モジュールは、ループ内の2つのアサインメントが組み合わせられ得ることを発見する。

【0165】

【数32】

Define ID1 ("main")

Declare ID2 ("x")

x := 0(#8)

While (x < 10(#8))

x := (x << 1) + 1(#8)

40

【0166】最終的に、HDL生成器モジュールは、統合用のVHDL RTLで書かれた以下の出力を生成する。初めにエントリ宣言があり、これにより、周辺とのインターフェースが記述される。

【0167】

【数33】

25

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use work.bach_std.all;
entity main is
    port(
        clk : in std_logic;
        reset : in std_logic;
        start : in std_logic;
        finish : out std_logic
    );
end main;

```

26

【0168】第2に、アーキテクチャがあり、これにより、エントリの挙動が記述される。これは、3つの部分、即ち、いくつかのローカル宣言と、制御パスのための状態マシンと、格納場所及びデータパスのためのレジスタ定義とに分かれる。

【0169】ローカル宣言

【0170】

【数34】

10

```

type sm_main_O_type is (state_main_O_0, state_main_O_1,
    state_main_O_3, state_main_O_4);
signal sm_main_O : sm_main_O_type;
signal var_x3 : unsigned(7 downto 0);
signal varw_x3 : unsigned(7 downto 0);
begin

```

【0171】メイン(main)のアーキテクチャRTLを、
以上に示す。

【0173】

【数35】

【0172】制御パス状態マシンは、

```

main_0: process (clk)
begin
    if ((clk'event and clk='1')) then
        if ((reset = '1')) then
            sm_main_0 <= state_main_0_0;
        else
            case (sm_main_0) is
                when state_main_0_0 =>
                    if ((start = '1')) then
                        sm_main_0 <= state_main_0_1;
                    end if;
                when state_main_0_1 =>
                    sm_main_0 <= state_main_0_3;
                when state_main_0_3 =>
                    if ((var_x3 >= unsigned('00001010'))) then
                        sm_main_0 <= state_main_0_0;
                    elsif ((var_x3 < unsigned('00001010'))) then
                        sm_main_0 <= state_main_0_4;
                    end if;
                when state_main_0_4 =>
                    sm_main_0 <= state_main_0_3;
            end case;
        end if;
    end if;
end process;

```

【0174】のように示される。

【0176】

【0175】格納場所及びデータパスは、

【数36】

27

```

var_x3 <= varw_x3;
proc_var_x3: process (clk)
begin
    if ((clk'event and clk='1')) then
        if ((reset = '1')) then
            varw_x3 <= conv_unsigned(0, 8);
        else
            if ((sm_main_0=state_main_0_4)) then
                varw_x3 <= conv_unsigned(((var_x3 * 2)
                    + unsigned ' ("00000001")), 8);
            elsif ((sm_main_0 =state_main_0_1)) then
                varw_x3 <= conv_unsigned(unsigned ' ("00000000"), 8);
            end if;
        end if;
    end if;
end process;
finish <= boo12bit((sm_main_0=state_main_0_0));
end RTL;

```

28

【0177】のように示される。

【0178】

【発明の効果】以上に説明したように、本発明によれば、集積回路の設計にあたって、C言語に類似した言語によって記述された集積回路の機能や仕様などに関するソフトウェアアルゴリズムが、高レベルの最適化を行う適切なコンパイラの使用によって、自動的に或いは半自動的に、ハードウェア記述言語（HDL）に変換（コンパイル）される。ソフトウェアアルゴリズムを記述する言語としては、並列処理や同期通信を記述できる高レベル言語（例えば、並列C言語）を使用することができる。従って、本発明によれば、集積回路の設計にあたって、高レベル言語で記述されたソフトウェアレベルからハードウェアレベルへの変換時間が短縮され、ハードウェア開発の効率が向上する。

【図面の簡単な説明】

【図1】本発明のある実施形態の一部を構成するハードウェアコンパイラの構造を示す概略図である。

【図2A】図1のコンパイラによって制御パスがどのように統合されるのかを概略的に示す図であり、特に、開始及び終了時間が1つである単一の処理を説明するための図である。

【図2B】図1のコンパイラによって制御パスがどのように統合されるのかを概略的に示す図であり、特に、2つの処理をシーケンシャルに実行する方法を説明するための図である。

【図2C】図1のコンパイラによって制御パスがどのように統合されるのかを概略的に示す図であり、特に、複数の処理を同時に実行する方法を説明するための図である。

【図3】簡単なプログラム例を実行するための可能な回路例を示す図である。

【図4A】表現をどのようにエンコードするのかを説明するための図であり、特に、単一のR表現の場合を説明するための図である。

【図4B】表現をどのようにエンコードするのかを説明

するための図であり、特に、A及びBの表現から表現A+Bをどのように作成するのかを説明するための図である。

【図5A】コール・バイ・バリューフアンクションコールがどのように行われるのかを説明するための図である。

【図5B】単純変数がどのように読み出されるのかを説明するための図である。

【図5C】チャンネルがどのように読み出されるのかを説明するための図である。

【図6A】L表現がどのようにエンコードされるのかを示す図であり、特に、単一のL表現インターフェースを説明するための図である。

【図6B】L表現がどのようにエンコードされるのかを示す図であり、特に、どのようにL表現とR表現を組み合わせさせてアサインメントを生成するのかを説明するための図である。

【図7A】特定のL表現がどのようにエンコードされるのかを示す図であり、特に、単純変数或いはレジスタへの書き込みがどのように行われるのかを説明するための図である。

【図7B】特定のL表現がどのようにエンコードされるのかを示す図であり、特に、外部メモリへの書き込みがどのように行われるのかを説明するための図である。

【図7C】特定のL表現がどのようにエンコードされるのかを示す図であり、特に、チャンネル出力がどのように行われるのかを説明するための図である。

【図8】条件ステートメントをどのように作成するのかを説明するための図である。

【図9】ループステートメントをどのように作成するのかを説明するための図である。

【図10A】どのようにリソースを作成するのかを説明するための図であり、特に、単純変数をどのように作成するのかを説明するための図である。

【図10B】どのようにリソースを作成するのかを説明するための図であり、特に、チャンネルをどのように作成

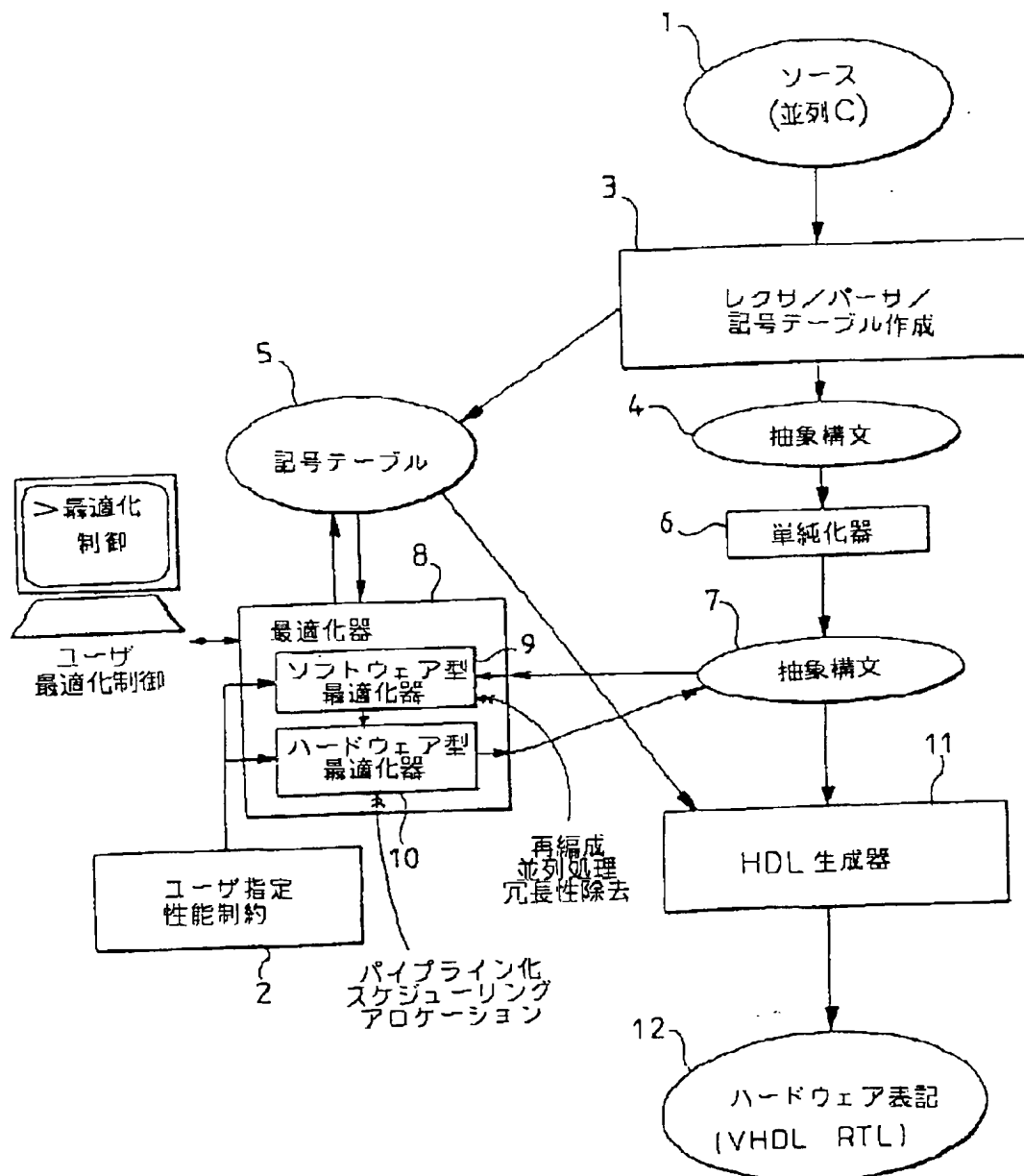
するのかを説明するための図である。

【符号の説明】

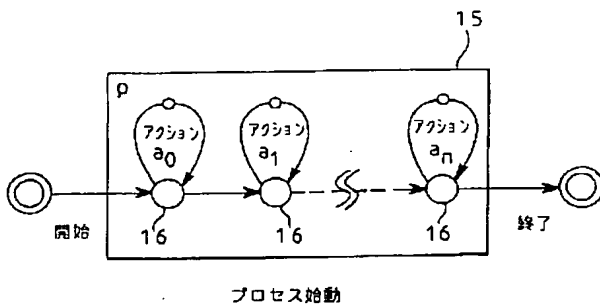
- 1 ソースコード
5 記号テーブル
6 単純化器モジュール

- 8 最適化器モジュール
9 ソフトウェア最適化器モジュール
10 ハードウェア最適化器モジュール
11 HDL生成器モジュール
12 ハードウェア表記（出力コード）

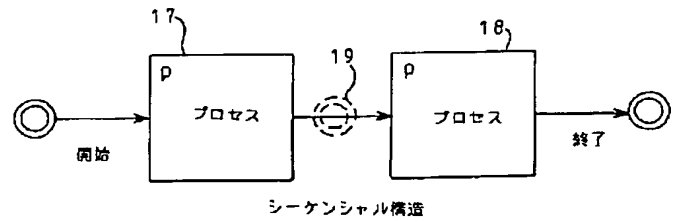
【図1】



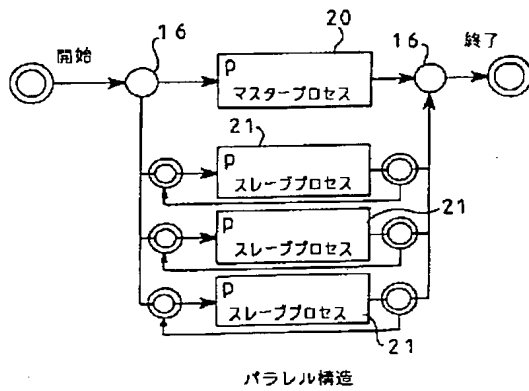
【図2A】



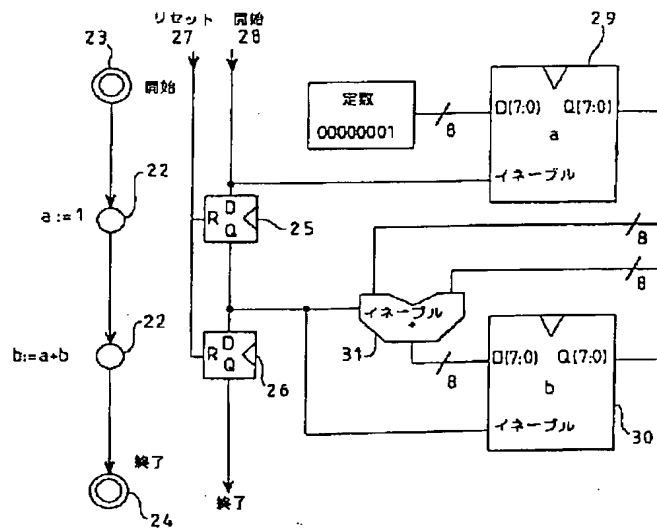
【図2B】



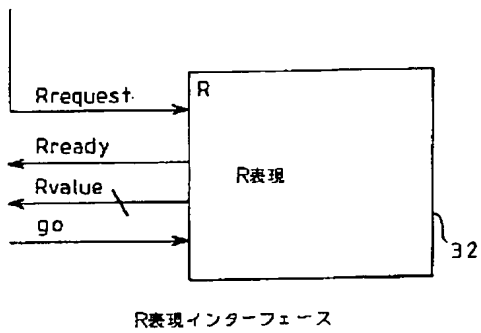
【図2C】



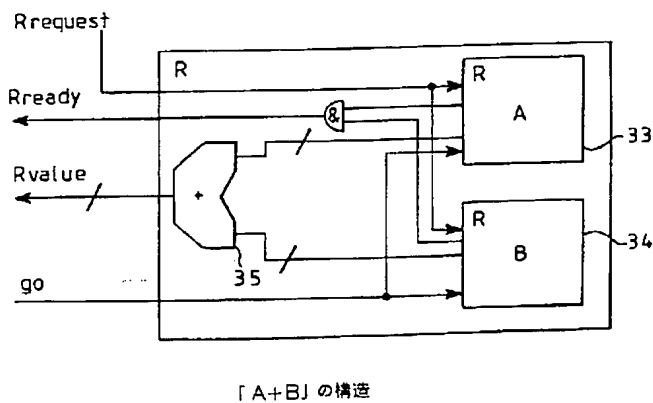
【図3】



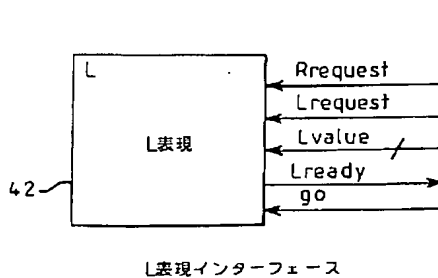
【図4A】



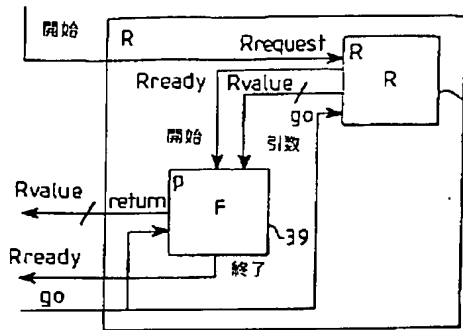
【図4B】



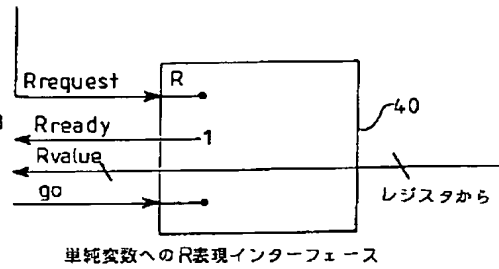
【図6A】



【図5A】

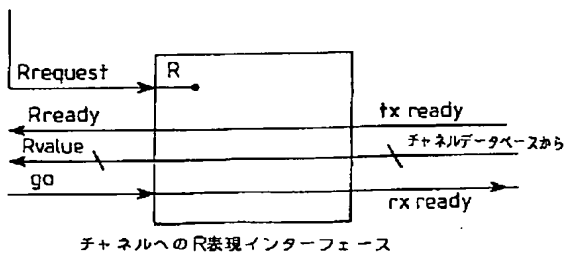
コール・バイ・バリュー関数 $f(R)$

【図5B】



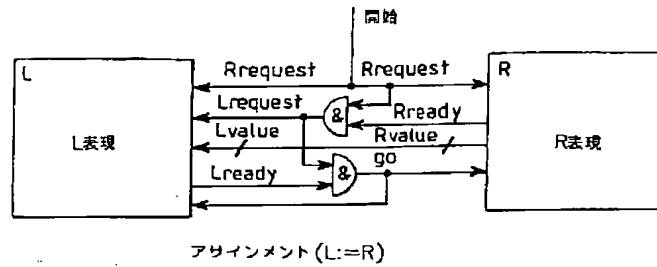
単純変数へのR表現インターフェース

【図5C】



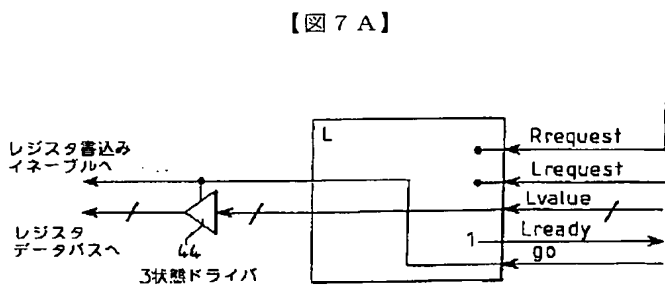
チャンネルへのR表現インターフェース

【図6B】



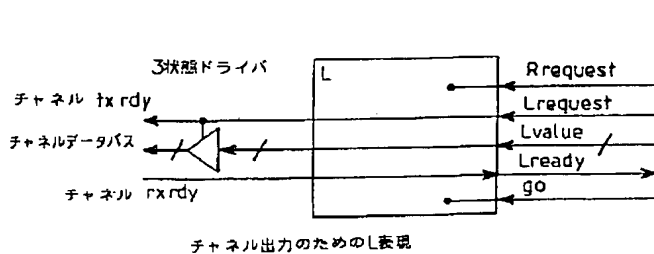
アサインメント (L:=R)

【図7B】



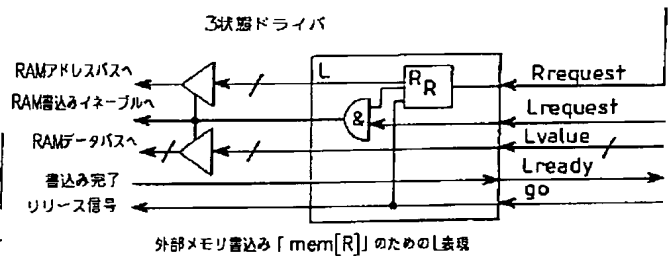
単純変数のためのL表現

【図7C】



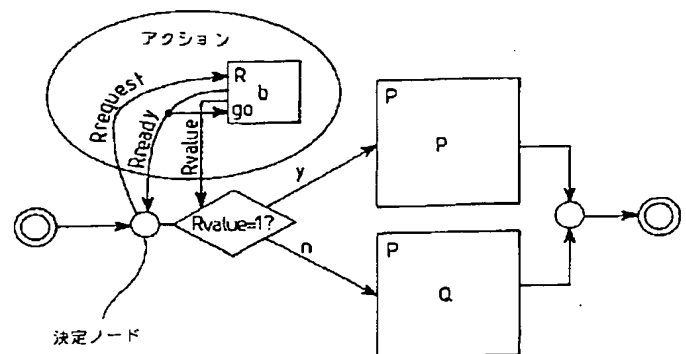
チャンネル出力のためのL表現

3状態ドライバ

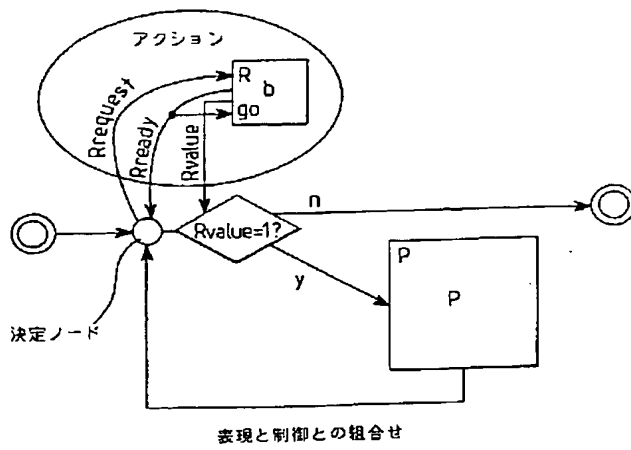


外部メモリ書き込み「mem[R]」のためのL表現

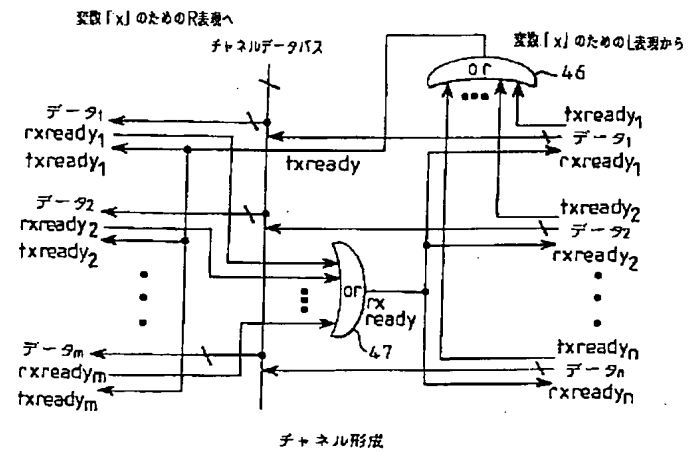
【図8】



【図 9】



【図 10 B】



【図 10 A】

